10 Utility Programs

This chapter describes the utility programs that accompany EnSight. The Server utility programs are located in `ENSIGHT_HOME/server/utilities` and the Client utility programs are located in `ENSIGHT_HOME/client/utilities` on the EnSight release tape or CD.

Utility programs are supplied on an "as is" basis and are unsupported. CEI will, however, try to assist in problem resolution.

Each utility program is presented below and accompanied with a brief overview that describes the function of the utility.

10.1 EnSight5 Programs

EnSight5 ASCII-to-Binary File Converter (asciitobin5)

The *asciitobin5* program runs on a Server host system to read ASCII EnSight 5.x files and convert them to C binary format files, which read much faster than ASCII files. Use this utility to facilitate the reading of large data files, especially when these files are read repeatedly.

EnSight Data Translation Library

The EnSight interface library ("libeio") provides a C API for reading and writing both the ASCII and the binary versions of the EnSight 5 format for geometry and results data. You can use it to simplify the process of writing translators or output modules for the EnSight format, as well as utilities that operate on the format.

Before using this library, you should be reasonably familiar with the EnSight 5 format described in section 2.5 of the EnSight User Manual.

The library (C source) can be found in the $ENSIGHT6_HOME/server/translators/libeio directory in your EnSight distribution. A translator for the unstructured "FAST" format that makes use of libeio can be found in $ENSIGHT6_HOME/server/translators/unf.

EnSight provides both ASCII and binary versions of its native data format. There are really only two reasons to use the ASCII format: if you need to actually look at the files or if you need to move a dataset to a computer system with a different binary format for numbers. Always use the binary format if possible. Not only does the I/O occur much faster, but the files will be smaller and will load into EnSight much faster as well.

You specify ASCII or binary output via the `SetFileType()` call. By default, output is set to binary.

Building the library

This library has been compiled and tested (to a limited extent) on the following systems:

SGI IRIX 4.0.5
SGI IRIX 5.3
HPUX 9.0.5
Solaris 2.3

1. Edit the Makefile for your system (as shipped, it is configured for IRIX).

2. To build the library, type "make". If you are porting the library to a new platform or operating system release, you may have to make some minor modifications to the Makefile and/or the source code.

3. To build executables that call routines in the library either include "`../libeio.a`" in your final link command or add the "`-leio`" option to the link command (which assumes the linker knows where to find the library).

Warnings

The following caveats apply to this initial release of libeio:

1. Error checking is a little skimpy at this point. It needs to be improved, especially for the input routines. In general, the input routines assume that a correctly formatted EnSight file is being read.

2. The input routines will only handle C binary files – not Fortran!

Hints

When reading EnSight format files into EnSight, you have the option of whether to load all parts, all but the first part, or the first part only. You can sometimes take advantage of this and save loading time as well as memory on the EnSight Client if you can load all but the first part. In many 3D applications (particularly CFD) one part can contain all 3D elements of the computational domain. Other parts typically contain boundary or shell elements. Since you don't really need to look at a graphical representation of the computational domain (if you have a boundary representation), you can avoid its initial load and display on the Client by having the 3D computational domain part as the first part in the EnSight geometry file and using the "all but the first part" load option in EnSight.

Output Routines

```
void SetFileType(int type)
```

`SetFileType()` sets the output type for subsequent calls to I/O routines. The type parameter is either `ASCII` or `BINARY` (as defined in eio.h). NOTE: the `ReadGeometry()` and `ReadParticleGeometry()` input routines will reset the type based on the type of the file last read.

The output routines are divided into two types: those that operate on the EnSight-based data structures (defined in eio.h) and those that accept raw arrays for output. The first four routines operate on the defined data structures:

```
int WriteGeometry(char *filename,Geometry *geo)
int WriteParticleGeometry(char *filename,ParticleGeometry *geo);
int WriteScalar(char *filename,Scalar *scl)
int WriteVector(char *filename,Vector *vec)
```

These routines take a completed structure for the corresponding item and write it to the file specified by "`filename`". See the definitions for `Geometry`, `ParticleGeometry`, `Scalar`, and `Vector` in `eio.h` for more info.

The remaining routines accept raw arrays for output.

```
int WriteGeoHeader(char *filename,char *des1,char *des2,int nodeid,int elemid)
```

`WriteGeoHeader()` begins the process of geometry file output. The `des1` and `des2` parameters are description lines for the model. The `nodeid` and `elemid` parameters should be set to one of the defined constants (e.g. `ID_OFF` or `ID_ASSIGN`) in eio.h. `WriteGeoHeader()` should be followed by `WriteGeoCoords()`.

```
int WriteParticleGeoHeader(char *filename, char *des)
```

`WriteParticleGeoHeader()` begins the process of particle geometry file output. Although particle files have two description lines in the header, the second one is ALWAYS "particle coordinates". `WriteParticleGeoHeader()` should be followed by a call to `WriteGeoCoords()`. Note that particle files must always have coordinate IDs!

```
void WriteGeoCoords(int partcoords, int num, int *id, float *coords)
```

`WriteGeoCoords()` appends coordinates to the geometry file opened by the previous call to `WriteGeoHeader()`. If the `nodeid` parameter to WriteGeoHeader() was either `ID_GIVEN` or `ID_IGNORE` then the `id` pointer must point to a list of num integers. The `coords` parameter must point to a list of 3*num floats in order X1,Y1,Z1,X2,Y2,Z2,...,Xn,Yn,Zn. `WriteGeoCoords()` should be followed by a call to

`WriteGeoPart()`.
`WriteGeoCoords()` is also used to output particle coordinates (e.g. following a call to `WriteParticleGeoHeader()`). Be sure to set the `partcoords` parameter to True when writing particle coordinates!

```
void WriteGeoPart(char *line)
```

`WriteGeoPart()` begins the process of part definition. A part header will be output to the file opened in the previous call to `WriteGeoHeader()`. `WriteGeoPart()` must be followed by one or more calls to `WriteGeoElem()`.

```
void WriteGeoElem(int elemtype, int num, int *id, int *nd)
```

`WriteGeoElem()` outputs a set of elements of the same type to the current part (as defined by the most recent call to `WriteGeoPart()`). The `elemtype` parameter must be one of the types defined in eio.h (e.g. `HEXA8` or `QUAD4`). num is the number of elements to output. If the `elemid` parameter to `WriteGeoHeader()` was either `ID_GIVEN` or `ID_IGNORE` then the `id` pointer must point to a list of num integers containing element ID numbers. The nd pointer points to a list of N*num integers, where N is the number of nodes in the particular type of element (e.g. 8 for a `HEXA8` type). Node ordering is defined section 3.8.

You can call `WriteGeoElem()` as many times as you like between calls to `WriteGeoPart()` to define different element sets belonging to a particular part.

```
int WriteRawScalar(char *filename, char *descrip, char *varname, int num, float *data)
```

`WriteRawScalar()` writes a scalar variable to the file named `filename`. The `varname` parameter will be saved and used in a subsequent call to `WriteResults()`. `num` is the number of values. `data` is a pointer to `num` floating point values. The values must be ordered the same as the coordinates in the corresponding geometry file.

```
int WriteRawVector(char *filename, char *descrip, char *varname, int num, float *data)
```

`WriteRawVector()` writes a vector variable to the file named `filename`. The `varname` parameter will be saved and used in a subsequent call to `WriteResults()`. `num` is the number of values. `data` is a pointer to 3*`num` floating point values. The values must be ordered the same as the coordinates in the corresponding geometry file.

```
int WriteResults(char *filename, Result *rp)
```

`WriteResults()` will output an EnSight “results” file describing a complete geometry plus results dataset. `rp` points to a Result structure (defined in `eio.h`) containing the desired information.

Input Routines

The input routines read a particular type of EnSight file and load the contents to a structure defined in eio.h.

`Geometry *ReadGeometry(char *filename)`

The `ReadGeometry()` routine reads a complete geometry file and returns the various components in the Geometry structure. It returns `NULL` on error. `ReadGeometry()` will automatically determine if the file is ASCII or binary and will set the type for subsequent reads.

`ParticleGeometry *ReadParticleGeometry(char *filename)`

The `ReadParticleGeometry()` routine reads a complete particle geometry file and returns the various components in the `ParticleGeometry` structure. It returns `NULL` on error. `ReadParticleGeometry()` will automatically determine if the file is ASCII or binary and will set the type for subsequent reads.

`Scalar *ReadScalar(char *filename, int num)`

`ReadScalar()` will read a scalar file and return a pointer to a `Scalar` structure (or `NULL` on error). `num` must equal the number of values to read. `ReadScalar()` will assume the file type (ASCII or binary) is the same as that determined in `ReadGeometry()` (but you can override with a call to `SetFileType()`).

`Vector *ReadVector(char *filename, int num)`

`ReadVector()` will read a vector file and return a pointer to a `Vector` structure (or `NULL` on error). `num` must equal the number of values (nodes) to read, *i.e.* there should be 3*`num` floats in the file. `ReadVector()` will assume the file type (ASCII or binary) is the same as that determined in `ReadGeometry()` (but you can override with a call to `SetFileType()`).

`Result *ReadResults(char *filename)`

`ReadResults()` will read an EnSight results file and return the information in an allocated `Result` structure.

10.2 MPGS4 Programs

MPGS4 ASCII-to-Binary File Converter (asciitobin4)

The *asciitobin4* program runs on a Server host system to read ASCII MPGS 4 data files and convert them to binary files, which read much faster than ASCII files. Use this utility to facilitate the reading of large data files, especially when these files are read repeatedly. See also *asciitobin5* above.

MPGS4 File Concatenater-Transformer

The programs under the `cat_transform4` directory run on a Server host system and perform various concatenation and transformation operations on MPGS 4 dataset files. For example, the following two utility programs are included in this directory:

cat_mpgs concatenates two or more MPGS 4 data files.

tform_mpgs translates and rotates MPGS 4 data files.

MPGS4 Geometry File Debug Filter (filter4)

The *filter4* program runs on a Server host system to read an MPGS 4 geometry file (either ASCII or binary). After reading the file, you can perform queries to aid in debugging connectivity information. You are prompted for a solid number, after which *filter4* will print all known information for that solid. If *filter4* cannot read the data, there is probably a problem with the data formatting.

MPGS4 Min-Max Scalar Finder (minmaxs4)

The *minmaxs4* program runs on a Server host system to scan a set of MPGS 4 (multiple time step) *scalar* files, and print the minimum and maximum scalar information. See also *minmaxv4* below.

MPGS4 Min-Max Vector Finder (minmaxv4)

The *minmaxv4* program runs on a Server host system to scan a set of MPGS 4 (multiple time step) *vector* files, and print the minimum and maximum vector information. See also *minmaxs4* above.

MPGS4 Structured Mesh Generator (structmesh4)

The *structmesh4* program runs on a Server host system, and creates an MPGS 4 geometry file that contains a 3D (cube) structured mesh.

10.3 Movie.BYU Programs

Movie.BYU File Polygon Reducer (reducemovie)

The *reducemovie* program runs on a Server host system to read Movie.BYU geometry, and output a geometry file with shared face information removed. This program is especially useful when dealing with geometry files that were created from FEM solid elements.

Depending on how smart a FEM translator is, the faces shared between two solid elements might be described twice in the geometry file. If *reducemovie* finds two faces (polygons in the Movie.BYU file) that share the same node numbers, both polygons are removed because they are both interior faces and should not be visible to the observer (unless the geometry is clipped open using the Z-clipping planes).

Running a FEM geometry that has been created using solid elements through this filtering program can reduce the number of polygons in the model dramatically, thus speeding postprocessing.

10.4 Keyboard Macro Maker (macromake)

The *macromake* program runs on the Client host system and assigns a keyboard key to a prerecorded EnSight command file (or files). The macro key code and command file name(s) are updated in the `macro.define` file, which stores your macro definitions.

The command file(s) can contain any sequence of valid EnSight commands that will execute each time the macro key is pressed while running EnSight. You can assign one command file to a *repeatable* macro key—the contents of the command file plays as long as the macro key is depressed. Macros are currently limited to single key definitions.

See: How To Define and Use Macros.